```
// ClassCloner.java
// Copyright (c) 2004. Borland Software Corporation All Rights Reserved.
package com.borland.jbuilder.refactor.async;
import java.io.*;
import com.borland.primetime.vfs.*;
import com.borland.jbuilder.jam.*;
import com.borland.jbuilder.java.*;
import com.borland.jbuilder.jom.*;
import com.borland.jbuilder.node.*;
import com.borland.jbuilder.repository.*;
import com.sun.tools.javac.code.*;
import com.sun.tools.javac.code.Symbol.*;
public class ClassCloner {
  private JBProject project;
  private JomFile newClonedJomFile;
  private JamClass existingJamClass;
  private String newClassName;
  /**
   * Will clone a class from existingJamClass and then rename the packag
e, class
   * and constructors based on newClassName
   *
   * @param project JBProject
   * @param existingJamClass JamClass
   * @param newClassName String
   */
  public ClassCloner(JBProject project, JamClass existingJamClass, Strin
g newClassName) {
    this.project = project;
    this.existingJamClass = existingJamClass;
    this.newClassName = newClassName;
  }
  /**
   * Clones the new class from existingJamClass and the newClassName, pa
rses
   * the newly create class which has the side affect of putting it in t
he symbol
   * table and then returns the ClassSymbol from the symbol table
   *
   * @return ClassSymbol
   */
  public ClassSymbol getNewClassSymbol() {
    createNewClass();
    return MemberInjector.findClassSymbol(newClassName, project);
  }
  private void createNewClass() {
    if (newClonedJomFile == null) {
      //Get an url for the new class from it's name and using the projec
ts paths
      Url newUrl = JomUtil.getUrlFromClassName(project, newClassName);
      ClassEntry ce = project.getRepository().getClassEntry(existingJamC
lass.getName());
      Url classFileUrl = ce.getFile();
      //Will contain the source code for the existing class (directly or
 decompiled)
      String newSource = null;
      //Try to create by decompiling if the compiled class file exists
      if (classFileUrl != null && VFS.exists(classFileUrl)) {
        ClassStubSource stub = null;
        stub = new ClassStubSource(classFileUrl);
        newSource = stub.toString();
      }
      //Else copy the source content from the source file
      else {
        SourceEntry se = ce.getSource();
        if (se != null) {
          try {
            newSource = new String(VFS.getBuffer(se.getFile()).getConten
```

```
t());
            }
            catch (IOException ex) {
            }
          }
        }
        if (newSource == null) {
          return;
        }
        //Create a SourceInfo for newSource and set to newUrl
        SourceInfo si = project.getSourceInfoManager().get(newSource);
        si.setUrl(newUrl);
        //Create an instance of JomFile so we can fix up the class and pac
kage
        //names
        newClonedJomFile = JomFile.instance(project, si, newUrl);
        if (newClonedJomFile != null) {
          JomClass jomClass = newClonedJomFile.getClass(existingJamClass.g
etType());
          if (jomClass != null) {
            //Set's the new class name (will fix up the classname and cons
tructors)
            jomClass.setName(JavaNames.getClassName(newClassName));
          }
          String newPackageName = JavaNames.getPackageName(newClassName);
          String oldPackageName = JavaNames.getPackageName(existingJamClas
s.getName());
          //If the new class is in a different package than fix up the pac
kage statement
          if (!newPackageName.equals(oldPackageName)) {
            newClonedJomFile.getPackage().setPackageName(newPackageName);
            newClonedJomFile.addImport(new JomImport(oldPackageName + ".*"
));
          }
          //Now commit changes which also enters the symbols into the symb
ol table
          //for the new class
          newClonedJomFile.commitAndEnterSymbols(false);
        }
      }
    }
    /**
     * Removes the cloned class
     */
    public void removeClonedClass() {
      try {
        if (newClonedJomFile != null) {
          VFS.delete(newClonedJomFile.getUrl());
          newClonedJomFile = null;
        }
      }
      catch (IOException ex) {
      }
    }
}
// MemberInjector.java
// Copyright (c) 2004. Borland Software Corporation All Rights Reserved.
package com.borland.jbuilder.java;
import com.sun.tools.javac.code.Symbol.ClassSymbol;
import com.sun.tools.javac.code.Symbol.VarSymbol;
import com.sun.tools.javac.code.Symbol.MethodSymbol;
import com.borland.jbuilder.java.filter.FilterHelper;
import com.borland.primetime.node.Project;
import com.sun.tools.javac.jvm.ClassReader;
import com.sun.tools.javac.util.Name;
import com.sun.tools.javac.util.Name.Table;
import com.sun.tools.javac.comp.Check;
import com.sun.tools.javac.comp.Env;
```

```
import com.sun.tools.javac.util.*;
import com.sun.tools.javac.code.Type;
import com.sun.tools.javac.code.Symtab;
import com.borland.jbuilder.jam.JamMethodType;
import com.borland.jbuilder.jam.JamType;
import com.sun.tools.javac.code.Type.MethodType;
import com.borland.primetime.vfs.VFS;
public class MemberInjector {
  static public ClassSymbol findClassSymbol(String fullName, Project pro
ject) {
      synchronized (CompilerManager.instance(project).getManager(true).
          getContextLock()) { // Alway get the JSp one (for now.)
        return FilterHelper.lookupClass(fullName,
            CompilerManager.instance(project).getManager(true).
            getProjectContext()); // Alway get the JSp one (for now.)
      }
  }
  static public void removeClassSymbol(String fullName, Project project)
 {
      synchronized (CompilerManager.instance(project).getManager(true).
          getContextLock()) { // Alway get the JSp one (for now.)
        Name.Table names = Name.Table.instance(CompilerManager.instance(
project).
            getManager(true).getProjectContext()); // Alway get the JSp
one (for now.)
        Name name = names.fromString(fullName);
        ClassReader.instance(CompilerManager.instance(project).getManage
r(true).
            getProjectContext()).classes.remove(name); // Alway get the
JSp one (for now.)
        Check context = Check.instance(CompilerManager.instance(project)
.
            getManager(true).getProjectContext());
        context.compiled.remove(name); // Alway get the JSp one (for now
.)
      }
  }
  static public VarSymbol findFieldSymbol(ClassSymbol owner, String name
, Project project, Env env) {
    try {
        synchronized (CompilerManager.instance(project).getManager(true)
.
            getContextLock()) { // Alway get the JSp one (for now.)
          Context context = CompilerManager.instance(project).getManager
(true).
              getProjectContext(); // Alway get the JSp one (for now.)
          ClassSymbol csym = FilterHelper.lookupClass(owner.flatName(),
context);
          if (csym == null) {
            return null;
          }
          return FilterHelper.lookupVar(csym, name, context, env);
        }
    }
    catch (Exception ex) {
      return null;
    }
  }
  static public VarSymbol addFieldSymbol(ClassSymbol owner, long flags,
                                         String varName, String sType,
                                         Project project) {
    assert owner != null;
    try {
        synchronized (CompilerManager.instance(project).getManager(true)
.
            getContextLock()) { // Alway get the JSp one (for now.)
          Type type = getTypeFromString(sType, project);
          assert type != null;
```

```
          VarSymbol ret = null;
          Name.Table names = Name.Table.instance(CompilerManager.instanc
e(
              project).
              getManager(true).
              getProjectContext()); // Alway get the JSp one (for now.)
          Name name = names.fromString(varName);
          ret = new VarSymbol(flags, name, type, owner);
          owner.members().enter(ret);
          return ret;
        }
    }
    catch (Exception ex) {
      return null;
    }
  }
  static public MethodSymbol findMethodSymbol(ClassSymbol owner,
                                              String name, String retTyp
e,
                                              String[] parameterTypes,
                                              Project project, Env env)
{
    assert owner != null;
    try {
        synchronized (CompilerManager.instance(project).getManager(true)
.
            getContextLock()) { // Alway get the JSp one (for now.)
          Context context = CompilerManager.instance(project).getManager
(true).
              getProjectContext(); // Alway get the JSp one (for now.)
          JamType[] paramTypes = new JamType[parameterTypes.length];
          for (int i = 0; i < paramTypes.length; i++) {
            paramTypes[i] = JamType.fromText(parameterTypes[i]);
          }
          String signature = JamMethodType.createSignature(JamType.fromT
ext(
              retType), paramTypes);
          MethodSymbol ret = FilterHelper.lookupFun(owner, name, signatu
re,
              context, env);
          return ret;
        }
    }
    catch (Exception ex) {
      return null;
    }
  }
  static public MethodSymbol addMethodSymbol(ClassSymbol owner, long fla
gs,
                                             String methodName, String r
etType,
                                             String[] parameterTypes,
                                             String[] thrownTypes,
                                             Project project) {
    assert owner != null;
    try {
        synchronized (CompilerManager.instance(project).getManager(true)
.
            getContextLock()) { // Alway get the JSp one (for now.)
          Type resType = getTypeFromString(retType, project);
          assert resType != null;
          Context context = CompilerManager.instance(project).getManager
(true).
              getProjectContext(); // Alway get the JSp one (for now.)
          MethodSymbol ret = null;
          Name.Table names = Name.Table.instance(context);
          Name name = names.fromString(methodName);
          ListBuffer paramTypes = new ListBuffer();
```

```
          ListBuffer thrTypes = new ListBuffer();
          for (int i = 0; i < parameterTypes.length; i++) {
            paramTypes.append(getTypeFromString(parameterTypes[i], proje
ct));
          }
          for (int i = 0; i < thrownTypes.length; i++) {
            thrTypes.append(getTypeFromString(thrownTypes[i], project));
          }
          MethodType mType = new MethodType(paramTypes.toList(), resType
,
              thrTypes.toList(), owner.type.tsym);
          ret = new MethodSymbol(flags, name, mType, owner);
          owner.members().enter(ret);
          return ret;
        }
    }
    catch (Exception ex) {
      return null;
    }
  }
  private static Type getTypeFromString(String sType, Project project) {
      Type type = null;
      Symtab syms = Symtab.instance(CompilerManager.instance(project).ge
tManager(true).getProjectContext()); // Alway get the JSp one (for now.)
      if (sType.equals("int")) { // NORES
        type = syms.intType;
      }
      else if (sType.equals("byte")) { // NORES
        type = syms.byteType;
      }
      else if (sType.equals("char")) { // NORES
        type = syms.charType;
      }
      else if (sType.equals("short")) { // NORES
        type = syms.shortType;
      }
      else if (sType.equals("long")) { // NORES
        type = syms.longType;
      }
      else if (sType.equals("float")) { // NORES
        type = syms.floatType;
      }
      else if (sType.equals("double")) { // NORES
        type = syms.doubleType;
      }
      else if (sType.equals("boolean")) { // NORES
        type = syms.booleanType;
      }
      else if (sType.equals("void")) { // NORES
        type = syms.voidType;
      }
      else if (sType.indexOf("[") != -1) { // NORES
        type = syms.arraysType;
      }
      else {
        type = findClassSymbol(sType, project).type;
      }
      return type;
  }
}
// refactor_1_0.dtd
// Copyright (c) 2004. Borland Software Corporation All Rights Reserved.
<?xml version="1.0" encoding="UTF-8" ?>
<!ELEMENT refactor ( (rename-package | change-method-signature | rename-
method | rename-field | rename-class)* ) >
<!ELEMENT declaring-class-name ( #PCDATA ) >
<!ELEMENT rename-method ( declaring-class-name, declared-method-signatur
e, old-method-name, new-method-name ) >
```

```
<!ATTLIST rename-method creation-time CDATA #REQUIRED >
<!ATTLIST rename-method id CDATA #REQUIRED >
<!ELEMENT declared-method-signature ( #PCDATA ) >
<!ELEMENT old-method-name ( #PCDATA ) >
<!ELEMENT new-method-name ( #PCDATA ) >
<!ELEMENT rename-class ( old-class-name, new-class-name ) >
<!ATTLIST rename-class creation-time CDATA #REQUIRED >
<!ATTLIST rename-class id CDATA #REQUIRED >
<!ELEMENT old-class-name ( #PCDATA ) >
<!ELEMENT new-class-name ( #PCDATA ) >
<!ELEMENT rename-package ( old-package-name, new-package-name ) >
<!ATTLIST rename-package creation-time CDATA #REQUIRED >
<!ATTLIST rename-package id CDATA #REQUIRED >
<!ELEMENT new-package-name ( #PCDATA ) >
<!ELEMENT old-package-name ( #PCDATA ) >
<!ELEMENT rename-field ( declaring-class-name, old-field-name, declared-
field-type, new-field-name ) >
<!ATTLIST rename-field creation-time CDATA #REQUIRED >
<!ATTLIST rename-field id CDATA #REQUIRED >
<!ELEMENT old-field-name ( #PCDATA ) >
<!ELEMENT declared-field-type ( #PCDATA ) >
<!ELEMENT new-field-name ( #PCDATA ) >
<!ELEMENT change-method-signature ( declaring-class-name, declared-metho
d-name, old-method-signature, new-return-type?, new-parameters? ) >
<!ATTLIST change-method-signature creation-time CDATA #REQUIRED >
<!ATTLIST change-method-signature id CDATA #REQUIRED >
<!ELEMENT declared-method-name ( #PCDATA ) >
<!ELEMENT old-method-signature ( #PCDATA ) >
<!ELEMENT new-return-type ( #PCDATA ) >
<!ELEMENT new-parameters ( parameter+ ) >
<!ELEMENT parameter ( parameter-name, parameter-type, default-value?, ol
d-index, new-index ) >
<!ELEMENT parameter-name ( #PCDATA ) >
<!ELEMENT parameter-type ( #PCDATA ) >
<!ELEMENT default-value ( #PCDATA ) >
<!ELEMENT old-index ( #PCDATA ) >
<!ELEMENT new-index ( #PCDATA ) >
// SymbolCreator.java
// Copyright (c) 2004. Borland Software Corporation All Rights Reserved.
package com.borland.jbuilder.refactor.async;
import com.borland.primetime.util.*;
import com.borland.jbuilder.jam.*;
import com.borland.jbuilder.java.*;
import com.borland.jbuilder.node.*;
import com.sun.tools.javac.code.*;
import com.sun.tools.javac.code.Symbol.*;
public class SymbolCreator {
  private JBProject project;
  private ClassSymbol classSymbol;
  private ClassCloner classCloner;
  /**
   * Constructor when a class already exists. Will lookup the classsymbo
l
   * required for adding members
   *
   * @param project JBProject
   * @param existingClass JamClass
   */
  public SymbolCreator(JBProject project,
                       JamClass existingClass) {
    this.project = project;
    classSymbol = MemberInjector.findClassSymbol(existingClass.getName()
, project);
  }
  /**
   * Constructor when the class does not exist and needs to be created f
rom
   * a cloned class
```

```
   *
   * @param project JBProject
   * @param existingClass JamClass - this the class that will used to cl
one
   * the new class
   * @param newClassName String - the name of the new class once cloned
   */
  public SymbolCreator(JBProject project,
                       JamClass existingClass,
                       String newClassName) {
    this.project = project;
    classCloner = new ClassCloner(project, existingClass, newClassName);
    classSymbol = classCloner.getNewClassSymbol();
  }
  public void clearSymbols() {
    if (classCloner != null) {
      classCloner.removeClonedClass();
    }
    project.getCompilerManager().purgeCompilerContext();
  }
  /**
   * From methodName and methodType create a method symbol for the class
 passed
   * in the contructor
   *
   * @param methodName String
   * @param methodType JamMethodType
   * @return MethodSymbol
   */
  public MethodSymbol addMethodSymbol(String methodName,
                                      JamMethodType methodType) {
    classSymbol = getClassSymbol();
    String[] parameters = JamUtil.convertJamTypeToString(methodType.getP
arameters());
    String returnType = methodType.getReturnType().toText();
    MethodSymbol newMethod = MemberInjector.addMethodSymbol(classSymbol,
        Flags.PUBLIC,
        methodName,
        returnType,
        parameters,
        EmptyArrays.STRING_EMPTY_ARRAY,
        project);
    return newMethod;
  }
  /**
   * From fromFieldName and fieldType create a new VarSymbol for the cla
ss
   * paseed in the constructor
   *
   * @param fromFieldName String
   * @param fieldType JamType
   * @return VarSymbol
   */
  public VarSymbol addFieldSymbol(String fromFieldName, JamType fieldTyp
e) {
    classSymbol = getClassSymbol();
    VarSymbol newField;
    newField = MemberInjector.addFieldSymbol(classSymbol,
                                             Flags.PUBLIC,
                                             fromFieldName,
                                             fieldType.toText(),
                                             project);
    return newField;
  }
  /**
   * Get the class symbol found during construction of this class
   *
   * @return ClassSymbol
```

```
   */
  public ClassSymbol getClassSymbol() {
    return classSymbol;
  }
}
```